UNIVERSIDADE FEDERAL DE SANTA CATARINA
Departamento de Informática e Estatística

Augusto Vieira Ávila

PROGRAMAÇÃO FUNCIONAL E REATIVA APLICADA AO DESENVOLVIMENTO DE INTERFACES COM O USUÁRIO EM APLICAÇÕES PARA *WEB*

Florianópolis
2017

Augusto Vieira Ávila

PROGRAMAÇÃO FUNCIONAL E REATIVA APLICADA AO DESENVOLVIMENTO DE INTERFACES COM O USUÁRIO EM APLICAÇÕES PARA *WEB*

Trabalho de Conclusão de Curso apresentado ao Departamento de Informática e Estatística da Universidade Federal de Santa Catarina como requisito parcial para obtenção do título de Bacharel em Sistemas de Informação.

Orientador: Prof. Leandro José Komosinski

Florianópolis

2017

RESUMO

O barateamento do *hardware* nos últimos anos fez com que o cliente de uma aplicação *web* - por exemplo, um *web browser* requisitando uma página *web* de um servidor - tenha grande capacidade de processamento devido ao uso de computadores e celulares mais robustos que os de anos atrás. Com isso, o modelo de aplicação cliente-servidor tem mudado com o passar dos anos. Antes, o servidor era responsável por processar toda a lógica da aplicação, e o cliente basicamente servia como um visualizador de documentos HTML. Atualmente, o computador de usuários comuns dispõe de tamanha capacidade computacional que a lógica da aplicação está migrando para o lado do cliente. Esta mudança fez com que aumentasse a complexidade para se desenvolver o lado do cliente da aplicação, pois agora recursos como processamento concorrente e requisições assíncronas são utilizados a todo instante.

O modelo atual de desenvolvimento de *software*, que em grande parte segue o paradigma de programação Orientado a Objetos, tem demonstrado não ser ideal para resolver problemas onde há um grande número de eventos assíncronos na aplicação, que é uma característica do *front-end* de aplicações *web*. Por meio de levantamento teórico da técnica de Programação Funcional e Reativa (PFR) e o estudo de tecnologias disponíveis atualmente no mercado, o presente trabalho demonstra alguns benefícios que a PFR traz para resolver este tipo de problema.

Ao final deste trabalho observou-se que apesar da PFR resolver alguns problemas inerentes ao desenvolvimento do *front-end* de aplicações para *Web*, esta técnica de programação exige um longo período de aprendizado por quem irá utilizá-la. Além disso, foram encontradas poucas referências para um aprofundamento prático neste tema.

Palavras-chave: Programação Funcional Reativa. Programação Funcional. Programação Reativa. Fluxo de Dados. *Web* 2.0. *Front-end*.

LISTA DE FIGURAS

LISTA DE TABELAS

LISTA DE ABREVIATURAS E SIGLAS

CSV Comma-separated values
DOM Document Object Model
GUI Graphical User Interface
JSON JavaScript Object Notation
OO Orientação a Objetos
PDV Ponto de Venda
PF Programação Funcional
PFR Programação Funcional e Reativa
REST Representational State Transfer
URI Uniform Resource Identifier
CSS Cascading Style Sheets
HTML HyperText Markup Language
HTTP Hypertext Transfer Protocol
RFC Request For Comments
API Application Programming Interface
CRUD Create, Read, Update e Delete
XML eXtensible Markup Language
SQL Structured Query Language
LINQ Language Integrated Query

SUMÁRIO

1 INTRODUÇÃO

Com a evidente evolução computacional do *hardware* no passar dos anos houve uma considerável queda no seu preço. O que antes apenas grandes empresas, centros de pesquisa de universidades e governos possuíam computadores, hoje o cenário é completamente diferente. A popularização de equipamentos como computadores, celulares e *tablets* é consequência da diminuição do custo deste tipo de *hardware*. Desta forma, novas oportunidades no avanço do desenvolvimento de aplicações, como é o caso de aplicações *web*, surgiram em razão deste fato.

Agora é possível realizar processamento de dados que antes não era viável. Atualmente, é notório o aumento de SPAs (*Single Page Application* - aplicação de uma única página) e da quantidade de estado que o *front-end* deve armazenar, realizar requisições assíncronas ao servidor da aplicação, salvar e gerenciar bases de dados local, *cache* de dados, entre outros (REDUX, 2016).

Esse tipo de aplicação é concebido em sua grande maioria em uma arquitetura de cliente-servidor. A camada cliente, também chamada de *front-end*, é a camada da aplicação que permite a interação do usuário com o *software*. Esta parte da aplicação é executada através de um navegador como o Google Chrome, Firefox, Internet Explorer e ganhou mais a atenção dos desenvolvedores pois agora é possível aproveitar o recurso de *hardware* (computador, celular, *tablet*, por exemplo) do usuário para processar uma maior quantidade de dados. Isso pode ser observado, por exemplo, com aplicações de empresas como Google e Facebook, as quais desenvolvem produtos com alto grau de responsividade e fluxo de dados com os usuários. A evolução destas aplicações é o marco da *Web* 2.0, onde a criação do conteúdo online não é mais restrita às empresas desenvolvedoras, e sim estendida a toda e qualquer pessoa com acesso à internet (O'REILLY, 2016).

Com um número maior de funcionalidades implementadas no *front-end*, surgem problemas de complexidade na sua implementação como, por exemplo, o processamento assíncrono de dados, a responsividade da interface e o controle do fluxo contínuo de dados que entram no *front-end* por meio de inputs do usuário, como também pelo servidor da aplicação (*back-end)*. Em vista desta complexidade, o paradigma de PF vem sendo utilizado com frequência em *frameworks* para

explorar características deste paradigma como a imutabilidade dos dados e a decomposição do problema em partes menores, que facilitam o desacoplamento da aplicação. Este fato é evidenciado pelo crescente número de utilitários Javascript focados em PF, como é o caso das ferramentas Lodash (LODASH, 2017) e Ramda (RAMDA, 2017). Além da programação funcional, outro paradigma importante para a solução de problemas como estes mencionados anteriormente é a programação reativa. Esse paradigma aumenta o grau de abstração do código do *software*, permitindo, assim, um foco maior na interdependência de eventos que estão ligados à lógica da aplicação (MEDEIROS, 2014).

A união de características de ambos os paradigmas dá origem à técnica chamada de Programação Funcional e Reativa. Ela permite que o desenvolvimento seja orientado a eventos e modular. O *software* pode, também, ser representado em função das suas entradas (inputs), como um fluxo contínuo de dados (BLACKHEATH; JONES, 2015).

A seguir são levantados os problemas inerentes ao desenvolvimento do *front-end* conforme as técnicas tradicionais. É importante destacar quais são estes problemas para que se consiga, ao final deste trabalho, perceber os pontos positivos da PFR na resolução destes mesmos tipos de aplicações.

1.1 O PROBLEMA

As aplicações estão cada vez mais dependentes de eventos assíncronos. Isso é evidente em aplicativos como Facebook, Google Docs e WhatsApp onde a interação do usuário constantemente requer dados que não estão disponíveis localmente. Ou seja, ao longo do período em que o aplicativo está ativo ele deve, diversas vezes, se atualizar com dados novos que podem ter surgido no *back-end* da aplicação. E o processo inverso também ocorre frequentemente, como quando um usuário envia uma mensagem para outro usuário, ou quando ele submete uma nova publicação no Facebook. É sempre esperado que caso ocorra algum erro durante estes eventos o aplicativo consiga se recuperar e continue seu processamento corretamente, sem que haja *side effects* - erros que são propagados de forma não proposital para outras partes da aplicação - e que o usuário seja informado com uma mensagem adequada sobre o problema. Estas expectativas de

qualidade são comuns a tantas aplicações atualmente que foi publicado um documento chamado “O Manifesto Reativo”, em 2014, que descreve estas e outras características de sistemas classificados como reativos. Retirado deste documento, sistemas reativos são “robustos, mais resistentes, mais flexíveis e melhor posicionados para sustentar as demandas modernas” (BONÉR et al., 2014).

Para que se consiga oferecer uma navegação fluida e satisfatória aos usuários como descreve o documento “O Manifesto Reativo”, é necessário que o processamento de tarefas longas, como a de uma requisição AJAX ao *back-end*, seja feita paralelamente a outras atividades na aplicação. Este é o problema que grandes aplicações têm que resolver: como lidar com toda a complexidade de interfaces dinâmicas, requisições assíncronas sendo feitas com frequência, tratamento e recuperação de erros, e ainda assim garantir fluidez e manutenibilidade da aplicação? Como Blackheath e Jones (2015) dizem no seu livro a respeito do aumento da complexidade da aplicação com o passar do tempo: Tudo parecia estar bem. As funcionalidades não estavam todas prontas ainda, mas o desenvolvimento estava ágil. O chefe estava feliz, os usuários estavam impressionados e os investidores otimistas. E então, simplesmente o *software* se desintegrou. A qualidade caiu, assim como a velocidade de desenvolvimento, e os clientes ficaram insatisfeitos (Blackheath; Jones, 2015, p. 1, tradução própria). Mansilla (2015) diz que uma das razões deste aumento de complexidade é que ainda tenta-se resolver problemas que são essencialmente assíncronos com programação imperativa, e que é necessária uma nova abordagem para resolver este tipo de problema. É necessária uma forma de abstrair a complexidade do processamento assíncrono em um contexto que há grande volume, variedade e velocidade de produção de novos dados.

A seguir são apresentadas algumas das técnicas populares utilizadas para tratar eventos assíncronos, mas que em contextos de aplicações complexas como citado anteriormente têm pontos negativos à aplicação.

1.1.1 FUNÇÕES DE CALLBACK

Um *callback* é uma função (A) passada como parâmetro a uma função (B) que irá executar uma operação assíncrona. Quando (B) terminar seu processamento, ele irá chamar a função (A) com o resultado da operação. Funções

de callback são usadas para gerenciar fluxos assíncronos de dados, como por exemplo I/O, acesso a um banco de dados, ou uma *input* do usuário à aplicação (NETWORK, 2016).

Segundo Sergi Mansilla (2015) *callbacks* são facilmente compreendidos e se tornaram o modo padrão de tratar fluxos assíncronos em aplicações desenvolvidas em JavaScript. Porém, esta simplicidade, em aplicações complexas, traz consequências negativas à manutenção do código.

O grande problema com callbacks é o entendimento da ordem de execução do programa (CALLBACKHELL, 2016). Isto diminui a legibilidade do código e consequentemente também a sua manutenção. O exemplo da Figura 1, escrito em linguagem Javascript, demonstra o uso do *callback* para enviar uma mensagem (um texto escrito pelo usuário da aplicação, por exemplo) ao servidor da aplicação (*back-end*). Foram abstraídos detalhes de implementação desta comunicação entre o cliente e o servidor para que se mantenha simples o entendimento do exemplo.

Figura 1 – *Callback*

```
1   /*
2       Enviar uma mensagem para o backend da aplicação
3   */
4   function enviarMsg(msg, sucessCallback, errorCallback){
5
6       webClient.enviarMsg(msg, function(webClientSuccess){
7           sucessCallback(webClientSuccess);
8
9       }, function(webClientError){
10          errorCallback(webClientError);
11      });
12
13  }
14
15  // Mensagem que será enviada ao servidor
16  var minhaMsg = ...
17
18  enviarMsg(minhaMsg, function(result){
19      console.log("Webclient conseguiu enviar minha mensagem para o backend");
20
21  }, function(error){
22
23      /*
24          Webclient não conseguiu enviar minha mensagem para o backend
25          Como contingência, salvar a mensagem no banco de dados local
26      */
27      myDatabaseObjet.insert(minhaMsg, function(dbResultOk){
28          console.log("myDatabaseObjet inseriu com sucesso minha msg no banco de dados");
29
30      }, function(dbResultError){
31          /*
32              Ocorreu algum erro ao salvar a mensagem no bando de dados local
33              TODO: mostrar mensagem de erro ao usuário
34          */
35      });
36
37  });
38
```

Fonte: Elaborado pelo autor

Apesar de fácil compreensão, os *callbacks*, quando usados em grande quantidade dificultam o entendimento do código como visto acima. Confunde-se onde o código começa a ser executado e onde há definição de funções. Abaixo é descrita a ordem de execução deste exemplo:

- linha 16 - buscar a mensagem que foi inserida pelo usuário na interface
- linha 18 - chamada do método enviarMsg passando como parâmetro a mensagem
- linha 4 - caso ocorra erro no envio da mensagem para o *backend*, o *callback* da linha 10 será chamado

- linha 10 - no caso de erro o fluxo de execução é direcionado à linha 21
- linha 27 - tenta-se inserir a mensagem no banco de dados local do cliente da aplicação. Caso ocorra erro, o callback definido na linha 30 é ativado

Outro aspecto negativo da implementação de *callbacks* é que uma vez que um método assíncrono é invocado não há como mudar os parâmetros da requisição. Meijer (2012) deu como exemplo a este problema em sua publicação “Your Mouse is a Database”, na sessão “Crossing Futures and Callbacks”, o caso de um método chamado *void getPeople* que tem o objetivo de buscar uma *Stream* dos nomes das pessoas, a cada minuto, que se submeteram a uma campanha de marketing. Uma vez invocado o método, a cada minuto serão recebidos os nomes das pessoas.

Porém, supondo-se que depois de invocado o método *getPeople* interesse à aplicação apenas os primeiros 100 nomes, não há como passar esta informação (parâmetro) à requisição já solicitada. Esse problema ocorre porque depois que é chamado o método *getPeople* é retornado *void*, ou seja, a única alternativa do desenvolvedor é aguardar o resultado do processamento, e invocar novamente o método *getPeople* passando como parâmetro o filtro de 100 nomes das pessoas.

A ideia principal por trás dos mecanismos que visam remover o *callback* das aplicações é mover a definição das funções (*callbacks*) para outras partes da aplicação de forma que a leitura do código se torne linear, de cima para baixo. Desta forma, novos desenvolvedores que venham a refatorar o código não terão que ler os detalhes de implementação de todos os métodos para que compreendam o que o programa faz (CALLBACKHELL, 2016). A abordagem proposta pela Programação Funcional Reativa é uma alternativa a substituição do uso de *callbacks*.

1.1.2 PROMISES

De acordo com Mansilla (2015), *Promise* foi um recurso criado com o intuito de resolver os problemas relacionados ao *callback*, como foi visto na seção anterior. Assim que uma requisição assíncrona é feita, se ela tiver sido executada usando os mecanismos de *Promise*, ela irá retornar imediatamente um resultado provisório (uma *Promise*), que pode ser usado em qualquer parte da aplicação. Quando esta requisição assíncrona for resolvida, ou seja, quando ela retornar algo ou

simplesmente um erro, a *Promise* será ativada e o resultado estará disponível (MANSILLA, 2015).

Algumas das vantagens do uso de *Promise* são apresentadas abaixo:

1. Alinhamento de operações (Chaining)
2. Resolução única
3. Tratamento de exceções

Descrição da vantagem 1: Alinhamento de operações (*Chaining*)

É a possibilidade de alinhar chamadas assíncronas sem que seja necessário criar diversos níveis aninhados como é visto no uso de *callbacks*. Desta forma, a leitura do código se torna fluida e intuitiva pois segue uma sequência linear, de cima para baixo (DEVELOPERS, 2016).

Na Figura 2 é apresentado um exemplo de como três requisições aninhadas poderiam ser implementadas com o uso de *callbacks*, e logo após será demonstrado como este mesmo cenário poderia ser reescrito com *Promises*.

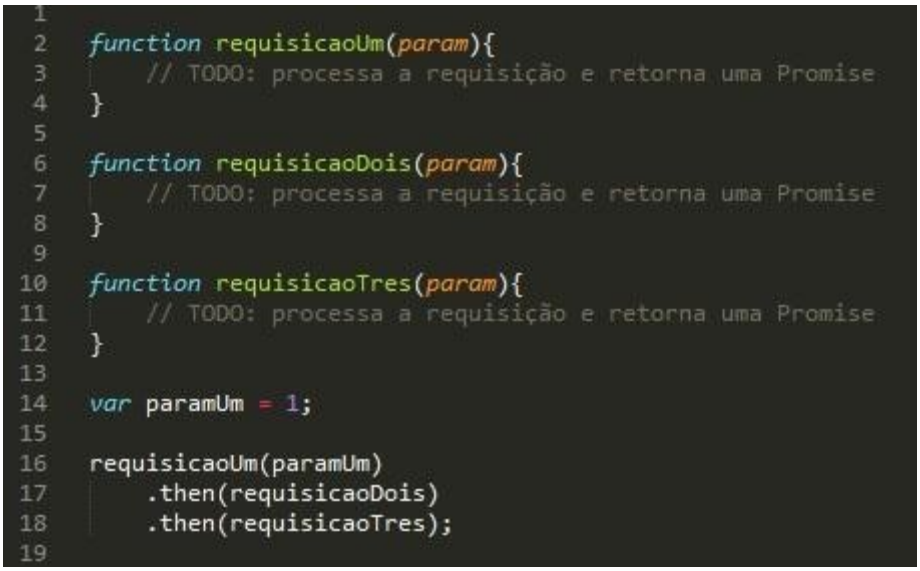
Figura 2 - Aninhamento de *callbacks*

```
var paramUm = 1;

servico.fazRequisicaoUm(paramUm, function(responseReqUm){

    var paramDois = 2;

    servico.fazRequisicaoDois(paramDois, function(responseReqDois){

        var paramTres = 3;

        servico.fazRequisicaoTres(paramTres, function(responseReqTres){

        });

    });

});
```

Fonte: Elaborado pelo autor

A Figura 3 apresenta uma possível implementação de como estas mesmas três requisições poderiam ser representadas com o uso de Promise.

Figura 3 – *Promise*



```
1
2   function requisicaoUm(param){
3       // TODO: processa a requisição e retorna uma Promise
4   }
5
6   function requisicaoDois(param){
7       // TODO: processa a requisição e retorna uma Promise
8   }
9
10  function requisicaoTres(param){
11      // TODO: processa a requisição e retorna uma Promise
12  }
13
14  var paramUm = 1;
15
16  requisicaoUm(paramUm)
17      .then(requisicaoDois)
18      .then(requisicaoTres);
19
```

Fonte: Elaborado pelo autor

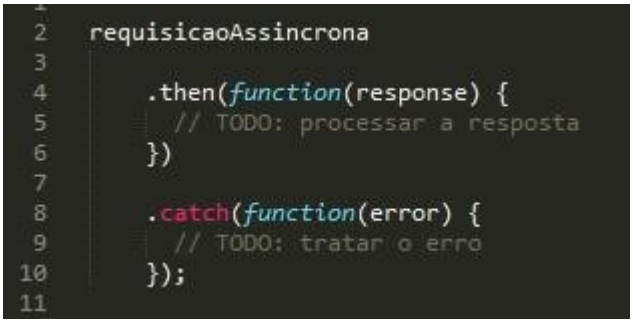
Descrição da vantagem 2: Resolução única

Com o uso de *callbacks*, não é garantida que a função que executa a operação assíncrona irá de fato executar o *callback* passado a ela uma única vez. Uma falha como esta pode introduzir bugs inesperados na aplicação. A *Promise* garante que a sua resolução, seja ela positiva ou negativa, ocorrerá apenas uma vez.

Descrição da vantagem 3: Tratamento de exceções

A *Promise* traz o conceito de *Try-Catch* para o tratamento de exceções. Este conceito se materializa, em *Javascript*, com a estrutura *catch* que pode ser adicionada ao final de uma chamada a uma *Promise* (DEVELOPERS, 2016).

Figura 4 – Promise com Try-Catch



```
requisicaoAssincrona

    .then(function(response) {
      // TODO: processar a resposta
    })

    .catch(function(error) {
      // TODO: tratar o erro
    });
```

Fonte: Elaborado pelo autor

Os exemplos mostrados acima das vantagens do uso de *Promise* foram desenvolvidos usando a linguagem *Javascript*. Porém, *Promise* é um conceito genérico e está disponível em inúmeras outras linguagens de programação.

1.1.3 ATUALIZAÇÃO DE ELEMENTOS DA GUI (GRAPHICAL USER INTERFACE) - PASSIVA

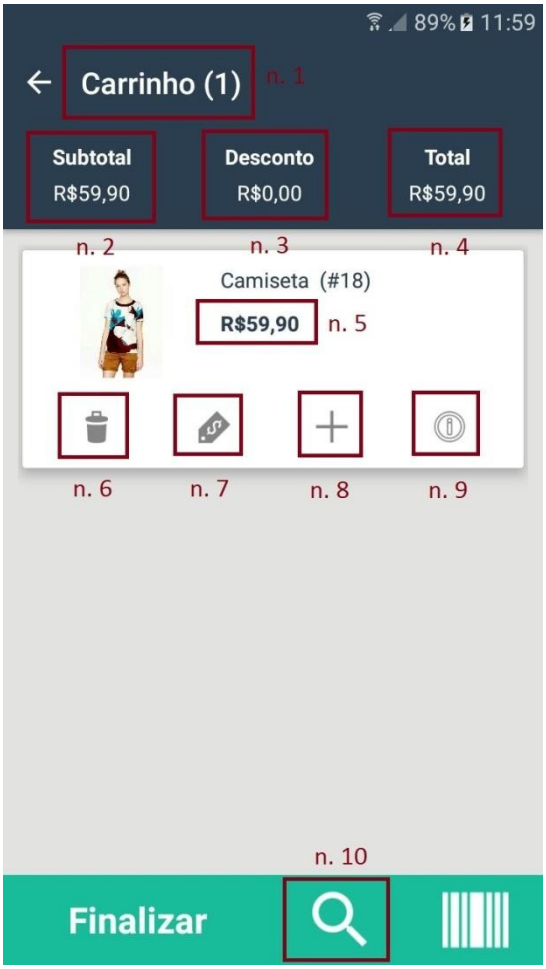
As chamadas GUIs (*Graphical User Interface*), em aplicações complexas, lidam com diversos eventos assíncronos que chegam à aplicação *front-end* tanto por meio de inputs do usuário, como também de dados oriundos do *back-end*. O principal problema é encontrar uma maneira de lidar com a mudança de estado quando um evento, como o clique do mouse sobre um botão, ocorrer na aplicação.

Um exemplo desta complexidade de atualização dos elementos da interface é uma aplicação de processamento de texto, como o Google Docs, da Google. Sempre que o usuário escreve uma palavra, é necessário que o programa recalcule o número total de palavras do documento, o número de parágrafos, reposicione a página atual caso não caibam mais palavras nela, salve o documento, propague o texto digitado para os outros colabores do documento, entre outras atualizações. Ou seja, muitas triggers são disparadas quando o evento “palavra_escrita” (um nome hipotético) ocorre.

Medeiros (2016), em sua publicação *Why Reactive Matters*, cita o problema relatado acima. Ele aborda duas possíveis soluções para esta questão: abordagem passiva, e abordagem reativa. Na Figura 5 é mostrado um típico cenário de atualização da interface do usuário de maneira que evidencie a abordagem passiva.

Segundo o autor, esta é a abordagem mais comum encontrada no mercado atualmente.

Figura 5 – Aplicação de PDV (Ponto de Venda) com destaques na interface



Fonte: Elaborado pelo autor

A Figura 5 representa a tela de venda de um PDV (Ponto de Venda) móvel. Nele é possível adicionar produtos ao carrinho, dar descontos para um produto, editar a quantidade de cada item, entre outras funcionalidades. Nesta imagem foram destacados com um retângulo vermelho os elementos visuais, são eles:

1. Mostra quantos itens têm adicionado no carrinho
2. Subtotal - total da venda sem os possíveis descontos de cada produto
3. Desconto - total de descontos da venda
4. Total - Subtotal menos os descontos

5. Preço de um item da venda
6. Remover produto do carrinho
7. Dar desconto para produto
8. Diminuir ou aumentar a quantidade de um produto
9. Visualizar mais informações do produto
10. Pesquisar um produto e adicioná-lo ao carrinho

Percebe-se que quando um produto é adicionado ao carrinho, os itens 1, 2, 3 e 4 devem ser atualizados. Utilizando a abordagem passiva da atualização da GUI, este cenário poderia ser implementado conforme mostrado na Figura 6:

Figura 6 – Carrinho de produtos

```
1   public class Carrinho{
2
3       List<Produto> produtos;
4       GUI gui;
5
6       public void addProduto(Produto novoProduto){
7           produtos.add(novoProduto);
8
9           // Atualizar os elementos da interface 1, 2, 3 e 4
10          gui.updateElemento1( this.produtos );
11          gui.updateElemento2( this.produtos );
12          gui.updateElemento3( this.produtos );
13          gui.updateElemento4( this.produtos );
14      }
15  }
16
```

Fonte: Elaborado pelo autor

Figura 7 – GUI

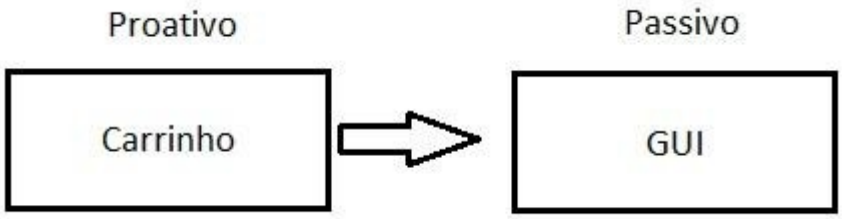
```
17  public class GUI{
18
19      Carrinho carrinho;
20
21      /*
22          Callback que é executado sempre que o botão de buscar produto (elemento 10)
23              for clicado
24      */
25      public void buscarProdutoOnClickListener(){
26          Produto produtoEncontrado = ... ;
27          carrinho.addProduto(produtoEncontrado);
28      }
29
30      /*
31          Contar quantos produtos estão adicionados ao carrinho e atualizar
32              a quantidade no elemento 1 da interface.
33      */
34      public void updateElemento1(List<Produto> produtos){
35          // TODO
36      }
37
38      /*
39          Percorrer todos os produtos do carrinho somando o valor de cada um
40              e atualizar o elemento 2 da interface com este valor.
41      */
42      public void updateElemento2(List<Produto> produtos){
43          // TODO
44      }
45
46      /*
47          Percorrer todos os produtos do carrinho somando o valor do desconto
48              de cada item e atualizar o elemento 3 da interface com este valor.
49      */
50      public void updateElemento3(List<Produto> produtos){
51          // TODO
52      }
53
54      /*
55          Percorrer todos os produtos do carrinho somando o valor do produto e
56              subtraindo seu desconto. Após isso, atualizar o elemento 4 da interface
57              com este valor.
58      */
59      public void updateElemento4(List<Produto> produtos){
60          // TODO
61      }
62  }
```

Fonte: Elaborado pelo autor

É importante notar na Figura 6 que a lógica de chamada aos métodos que atualizam a interface é de responsabilidade do método *addProduto*.

Esta é chamada de abordagem passiva pelo fato dos métodos de atualização da interface serem executados por qualquer parte do programa, permitindo que outros módulos do sistema modifiquem o seu estado, pois eles são públicos. Ou seja, estes segmentos de código de atualização “não sabem” quando devem ser executados, em quais contextos eles serão ativados.

Figura 8 – Abordagem passiva



Fonte: Elaborado pelo autor

A Figura 8 mostra que o Carrinho tem controle sobre quando o interface deve ser atualizada, e não o contrário. Ou seja, o módulo GUI são sabe da existência do módulo Carrinho ou de qualquer outro que venha a executar seus métodos públicos (CYCLE.JS, 2016).

1.2 OBJETIVO

O presente trabalho visa identificar como a técnica de PFR pode melhorar a qualidade do *software* relacionado ao desenvolvimento do *front-end* de uma aplicação para *web*.

1.3 METODOLOGIA

A metodologia do presente trabalho está segmentada em três etapas, são elas:

Etapa 1: Levantamento das técnicas que são utilizadas no desenvolvimento do *front-end* atualmente no mercado, e quais seus problemas.

Etapa 2: Pesquisa exploratória da técnica PFR através uma revisão bibliográfica.

Etapa 3: Nesta etapa é feita uma análise sobre os pontos positivos e negativos encontrados na técnica PFR em comparação com as técnicas atuais utilizadas para desenvolver o *front-end* de aplicações ricas em interações com o usuário.

1.4 ORGANIZAÇÃO DO TRABALHO

No capítulo II, Fundamentação Teórica, são apresentados os assuntos que são requisitos para o estudo da PFR. São eles: Arquitetura cliente-servidor (2.1), Programação Funcional (2.2) e *Stream* (2.3).

No capítulo III, Programação Funcional Reativa, é apresentada a definição desta técnica de programação, seguido de um exemplo com pequenos segmentos de código de forma que facilite o vínculo entre o conteúdo teórico desta técnica e como ela se materializada em forma de *software*. Ainda no capítulo III são apresentadas três tecnologias que utilizam como base a técnica PFR, e em seguida é feita uma análise comparativa entre elas.

2 FUNDAMENTAÇÃO TEÓRICA

Este capítulo apresenta os assuntos que são pré-requisitos para o entendimento da técnica de PFR. As subseções são: Arquitetura cliente-servidor (2.1), Programação Funcional (2.2) e *Stream* (2.3).

2.1 ARQUITETURA CLIENTE-SERVIDOR

Com a redução do custo do *hardware* durante a década de 1980, muitas organizações transitaram de uma computação de servidores centralizados, como mainframes, para uma com uso de *fat clients* (computadores com funcionalidades ricas, independente de um servidor central). Esta mudança de arquitetura fez aumentar a complexidade do desenvolvimento do *software* uma vez que agora ele deve funcionar de maneira distribuída, e novas tratativas de sincronização e comunicação entre as máquinas devem ser feitas (D'AMORE; OBERST; 1983). A arquitetura cliente-servidor pode ser utilizada em diversos cenários, como aplicações de e-mail, sistema de impressão via rede, e a *World Wide Web*. Haja vista o presente trabalho se tratar de uma técnica de programação que ajuda o desenvolvedor na camada cliente da aplicação, é necessário o entendimento de como a aplicação cliente e servidor se comunicam no contexto de aplicações *Web*.

2.1.1 DESENVOLVIMENTO FRONT-END EM APLICAÇÕES WEB

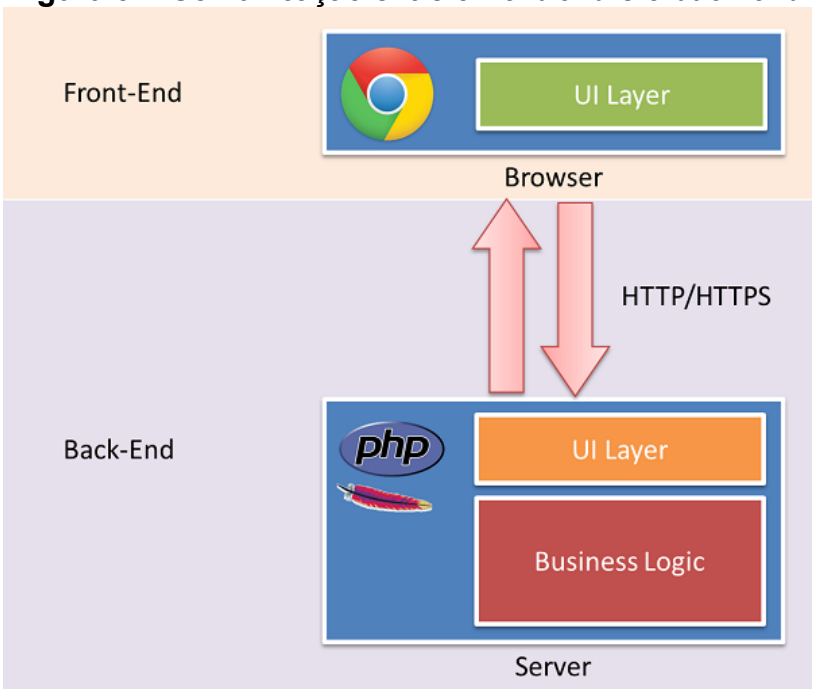
O *front-end* da aplicação é a camada responsável por interagir com o usuário. Ela deve permitir *inputs* do usuário, tratar estes dados de entrada conforme necessário, e após processá-los, apresentar o resultado na tela (*output*).

Nos dias de hoje é um desafio grande ser um desenvolvedor *front-end*. Há uma grande variedade de dispositivos nos quais os usuários querem acessar as aplicações, ora pelo computador, ora pelo celular ou tablet. Da mesma forma há uma variedade grande de versões de navegadores, o que dificulta a compatibilidade da aplicação, assim como entre diferentes modelos e fabricantes de dispositivos.

No contexto de aplicações *Web*, o desenvolvedor *front-end* trabalha basicamente com três tecnologias: HTML, CSS e Javascript. O HTML e CSS são as

tecnologias responsáveis por construir os elementos visuais da tela, como os botões, formulários, imagens, tabelas, entre outros. Já o Javascript é uma linguagem de programação responsável por deixar as páginas *Web* dinâmicas. Com a evolução desta linguagem e do poder de processamento dos computadores dos usuários é possível criar aplicações sofisticadas que vão muito além de simples regras de validação de formulários. Grande parte da regra de negócio da aplicação pode ser desenvolvida diretamente no *front-end*, e isso trás uma maior complexidade na sua concepção. A PFR é uma das técnicas disponíveis atualmente que ajuda o desenvolvedor *front-end* nesses desafios inerentes a esta camada da aplicação.

Figura 9 – Comunicação entre o *front-end* e o *back-end*



Fonte: https://www.nczonline.net/blog/2013/10/07/node-js-and-the-new-web-front-end/

2.1.2 DESENVOLVIMENTO BACK-END EM APLICAÇÕES WEB

O *back-end* da aplicação, no contexto de aplicações *Web*, é o responsável pelas regras de negócio, persistência dos dados dos usuários em bancos de dados, escalabilidade, entre outros. Enquanto o *front-end* é a camada que o usuário interage, o *back-end* normalmente é uma aplicação instalada em um servidor remoto, não acessível diretamente pelos usuários (ZAKAS, 2013).

As tecnologias utilizadas no desenvolvimento do *back-end* são diferentes das do *front-end*. Na Figura 9 acima foi dado como exemplo uma combinação de servidor Apache com a linguagem de programação PHP, e poderia ser utilizado também um banco de dados MySQL. Não é o objetivo deste trabalho entrar em detalhes sobre o conjunto de tecnologias disponíveis para uso no *back-end*, mas sim evidenciar as diferenças de responsabilidades entre estas camadas da aplicação. A seguir será apresentado como o *front-end* e *back-end* se comunicam, através do protocolo Hypertext Transfer Protocol (HTTP) e uso de Application Programming Interface (API).

2.1.3 PROTOCOLO HTTP (HYPERTEXT TRANSFER PROTOCOL)

O protocolo HTTP é um protocolo de aplicação responsável pela definição de regras e tratativas na comunicação entre uma aplicação cliente e uma servidora. Ele está em uso desde 1990 pelo *World-Wide Web* até os dias atuais. Segue abaixo a definição deste protocolo pela RFC 2616 (FIELDING et al., 1999):

> HTTP é um protocolo de aplicação para sistemas distribuídos, colaborativos, e de hipermídia. Ele é um protocolo genérico, sem estado, que pode ser utilizado além do seu uso em hipertextos, como servidores de nome e sistemas de gerenciamento de objetos distribuídos, através dos seus métodos de requisição, códigos de erro e cabeçalhos. (FIELDING et al., 1999, tradução livre).

Aplicações dinâmicas, como as redes sociais Facebook e Instagram, têm uma demanda alta por informações em tempo real. Neste tipo de aplicação é necessário que o *front-end* seja constantemente atualizado com o *back-end*. Para isso, a aplicação *front-end* faz requisições ao *back-end* em busca das informações mais atuais de forma que ele possa se atualizar e exibir o conteúdo na tela para o usuário. Essa comunicação é feita utilizando o protocolo HTTP.

O conteúdo na aplicação servidora é disponibilizado através de URIs. As URIs são caminhos específicos no servidor utilizados para disponibilizar recursos da aplicação servidora. Por exemplo, a página inicial do blog de uma pessoa pode ser esta: *http://www.meu-blog.com/home.html*. Percebe-se nesta URI que são

informados o protocolo utilizado (http), o domínio da aplicação (*www.meu-blog.com*) e o caminho para o recurso que se deseja obter (/home.html). Neste exemplo a aplicação servidora encontrará a página */home.html* e a retornará a quem iniciou a requisição.

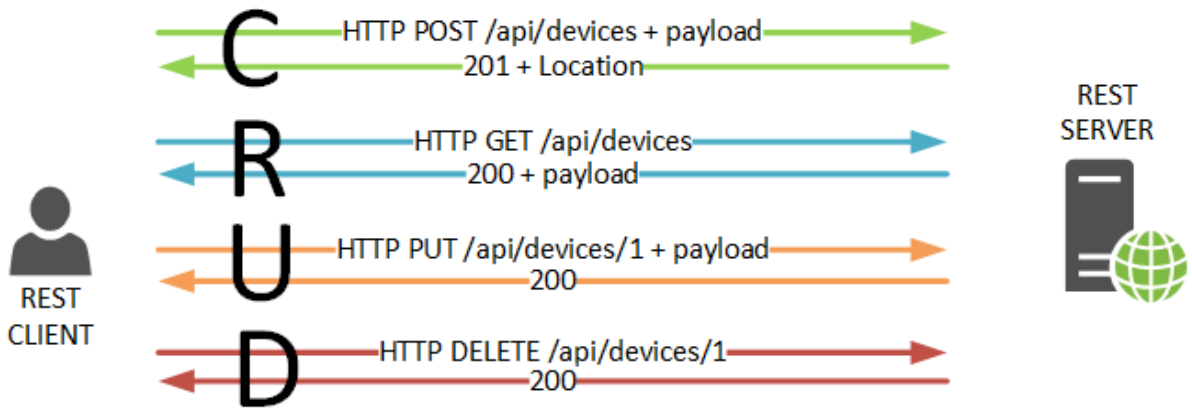
2.1.4 COMUNICAÇÃO VIA API REST

Assim como a GUI facilita a interação do usuário com a aplicação, as APIs simplificam o desenvolvimento da aplicação pelos programadores. API é o acrônimo para Application Programming Interface (Interface de Programação de Aplicativos), e é uma abordagem muito utilizada para comunicação entre as aplicações *front-end* e *back-end*, e utiliza como base o protocolo HTTP. O objetivo geral é que o *front-end* consiga utilizar recursos do *back-end*, sem que ele precise conhecer os detalhes de implementação do servidor (CLARKE, 2004). Por exemplo, uma aplicação de gerenciamento de arquivos remotos pode permitir que o usuário mova um arquivo de uma pasta para outra. As responsabilidades desta funcionalidade devem ser separadas entre ambas as camadas da aplicação. O *front-end*, por exemplo, pode requisitar um serviço de API do servidor que irá realizar especificamente esta tarefa. Já o servidor, ao receber a solicitação do cliente irá receber os parâmetros necessários do *front-end* e irá mover o arquivo de uma pasta para outra. Ou seja, o *front-end* desta aplicação não precisa conhecer os detalhes do sistema de arquivos e estrutura de diretórios do *back-end*, assim como também não precisa conhecer a lógica utilizada para mover um arquivo de um diretório a outro.

Um estilo de API que está sendo utilizado em larga escala são as APIs REST. O nome REST vem de *Representational State Transfer* (Estado de Transferência Representativo, tradução livre). As APIs REST são usadas para representar recursos do servidor e permitir realizar as ações CRUD (criar, ler, atualizar *e* deletar) através de URIs e o uso dos verbos do protocolo HTTP. Uma das vantagens do uso desse tipo de API é permitir o desacoplamento de tecnologias e prover uma interface bem definida de comunicação entre o *front-end* e *back-end*. Isso facilita o desenvolvimento e manutenção do *software* por equipes distintas, sem que os profissionais que trabalham no desenvolvimento do *front-end* tenham que conhecer os detalhes do banco de dados e linguagem de programação utilizados no *back-end* (FIELDING, 2000).

A Figura 10 mostra as quatro operações CRUD possíveis em uma API REST: *Create* (Criar), *Read* (ler), *Update* (atualizar) e *Delete* (Deletar). Ao lado esquerdo da imagem tem a representação de uma aplicação que fará uso dos recursos da API. Ao lado direito é a representação de uma aplicação *back-end* que provê seus serviços através de uma API REST. Como apresentado anteriormente, utilizam-se os verbos do protocolo HTTP para indicar ao servidor qual ação de deseja realizar no *back-end*.

Figura 10 - Operações CRUD em uma API REST



Fonte: http://networkop.co.uk/blog/2016/01/01/rest-for-neteng/

A Figura 10 mostra como exemplo a URI **/api/devices**. Esta URI é a representação de *device* (dispositivo) para o servidor. Em todas essas URIs o formato utilizado é o *JavaScript Object Notation* (*JSON)*. Este formato é simples para o ser humano criar e interpretar, assim como também tem baixo custo operacional para realizar o *parse* pela máquina.

Toda comunicação entre o *front-end* e *back-end* das aplicações citadas neste trabalho seguirão a mesma abordagem do exemplo acima. A aplicação cliente receberá *inputs* do usuário, e quando necessário, irá se comunicar com o servidor via APIs REST.

A próxima sessão deste trabalho apresentará o Cálculo Lambda e conceitos fundamentais do paradigma de Programação Funcional, que é um pré-requisito para o entendimento da Programação Funcional e Reativa.

2.2 PROGRAMAÇÃO FUNCIONAL

Para que seja possível o entendimento completo da técnica PFR, há necessidade de se ter conhecimentos mínimos sobre o paradigma de Programação Funcional.

O Cálculo Lambda é um sistema formal baseado na abstração e aplicação de funções. A sua versão não tipada (*untyped version*) foi proposta por Alonzo Church na década de 1930. Embora simples, este sistema é tão expressivo quanto a Máquina de Turing. O Cálculo Lambda teve grande influência no desenvolvimento de linguagens de programação que seguem o paradigma de programação funcional, como por exemplo: LISP, Ocaml, Haskell, entre outros (MACHADO, 2013).

Com linguagem sintática simples, o Cálculo Lambda visa representar problemas matemáticos a partir da combinação de funções. Além disso, por também ter semântica simples, passa a ser natural a representação de problemas computacionais e, por consequência, facilidade em verificar a sua corretude.

Uma característica marcante deste sistema formal é a definição de funções que recebem um único parâmetro (BARKLEY, 1984). Com isso, é possível especializar a definição de funções que viabilizam, desta forma, o seu reuso. Com a implementação de funções específicas, elas podem ser usadas como argumentos em funções de mais alto nível, também chamadas de *Higher Order Functions*.

As primeiras linguagens de programação foram desenvolvidas para controlar máquinas específicas. Ou seja, uma linguagem de programação poderia ser usada somente com a mesma arquitetura de máquina sob a qual a especificação da linguagem se embasou. Rapidamente percebeu-se dois grandes problemas. Primeiro, o uso da linguagem é restrito a apenas um tipo de arquitetura de máquina. Segundo, a representação dos problemas por meio da linguagem de programação não era de entendimento trivial, natural. A partir de então, uma série de linguagens foram desenvolvidas com o objetivo de resolver ambos os problemas acima, começando com Fortran, nos anos 50 (HUDAK, 1989).

Assim como Fortran, as primeiras linguagens de programação seguiam, na sua grande maioria, o paradigma imperativo de programação, onde os comandos na programação se aproximam do comportamento de uma máquina. Apesar do uso extensivo deste paradigma, uma abordagem matemática foi desenvolvida por Alonzo

Church em 1930, chamada de Cálculo Lambda que em 1950 veio a se concretizar como uma técnica de programação (HINSEN, 2009).

As linguagens de programação funcionais foram utilizadas na ciência da computação por décadas. Porém, o seu uso para desenvolver aplicações comerciais não era expressivo quando comparado ao paradigma imperativo. Segundo Hinsen (2009), são dois os motivos principais que justificam este fato: a dificuldade do seu aprendizado devido à completa mudança de pensamento necessária para implementar a aplicação; e a perda de eficiência de programas funcionais em comparação aos imperativos haja vista o segundo se aproximar da linguagem de máquina.

Apesar dos fatos citados anteriormente, o interesse por linguagens funcionais cresceu recentemente, como é visto, por exemplo, na incorporação de recursos funcionais em linguagens como F# (MICROSOFT, 2017) e Fortress (RISKE, 2005). Esta tendência pode ser explicada pelo também crescente uso da programação concorrente, utilizando múltiplos processadores, e pelo fato da PF permitir que o desenvolvedor trabalhe em um nível mais alto de programação que o imperativo, o que aumenta a robustez e traz uma maior facilidade na manutenção do *software* (HINSEN, 2009).

2.3 STREAM

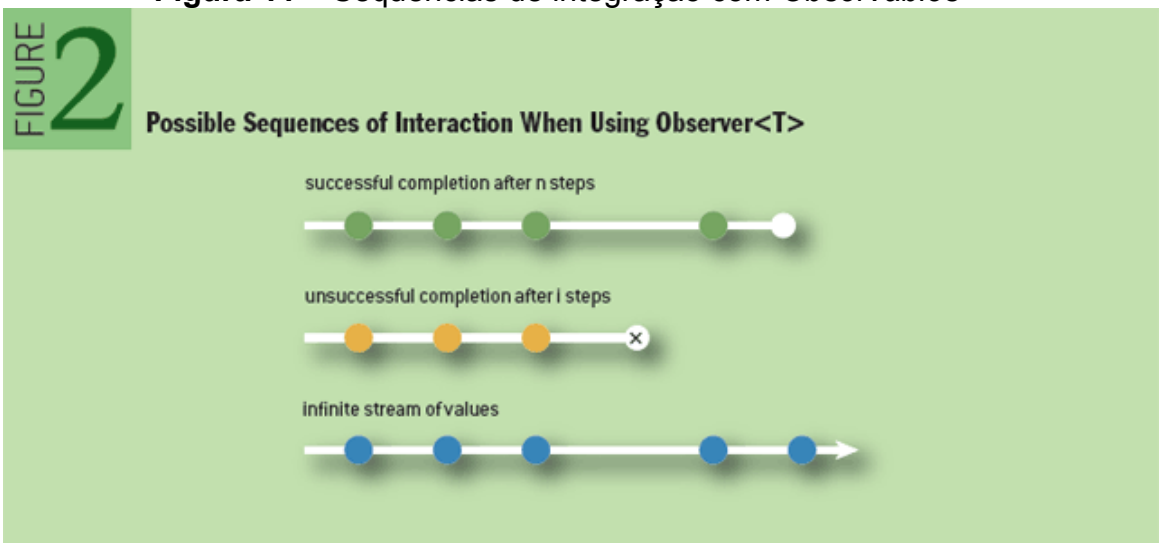
Outro conceito necessário para o entendimento deste trabalho é a chamada *Stream*, que é utilizada no domínio da computação de forma abrangente. Uma *Stream* é uma abstração de uma troca de dados entre um produtor e um consumidor, que representa um fluxo de dados potencialmente infinito. Conforme o dado é produzido pelo produtor, este dado é enviado ao consumidor. Ou seja, não há necessidade de armazenamento destes dados em estruturas auxiliares, como batches, para então serem transmitidos. Diversas são as suas aplicações, como por exemplo a leitura e escrita de um arquivo em uma base de dados, a comunicação *Socket* entre um servidor e um cliente, o envio de um arquivo de música entre dois celulares, entre outras (STELOVSKY; ASCHWANDEN, 2002).

Conforme a capacidade computacional dos equipamentos aumenta, há maior possibilidade de novas funcionalidades, mais complexas que as atuais, serem

desenvolvidas. Com isso, uma quantidade maior de informações dos usuários destas aplicações pode ser utilizada, como sensores de movimento, leitura óptica, sensores de temperatura e dispositivos de realidade virtual. Todas estas informações que são captadas dos usuários podem ser enviadas em forma de uma *Stream* de dados à aplicação, que por sua vez deve interpretar o contexto destes dados e processá-los, provendo outputs ao usuário (STELOVSKY; ASCHWANDEN, 2002).

Quando *Streams* são usadas no desenvolvimento de *software*, deve-se seguir uma nova abordagem de implementação uma vez que as atuais não satisfazem o requisito de trabalhar com dados possivelmente infinitos. A Figura 11 destaca três formas de receber dados através de uma *Stream*:

Figura 11 – Sequências de integração com *Observables*



Fonte: MEIJER (2012).

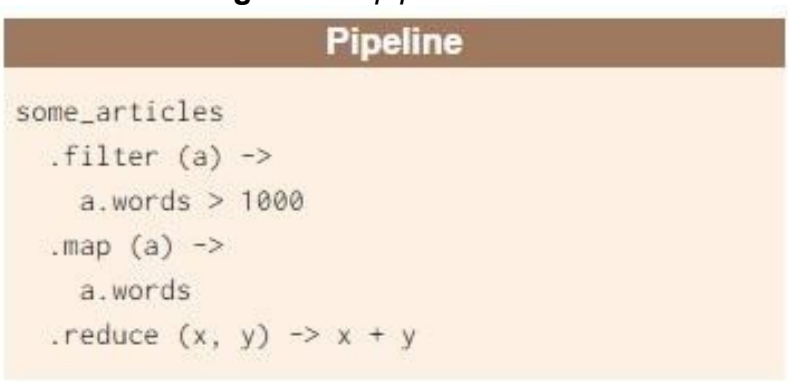
1. Após emitir uma sequência finita de dados a *Stream* é fechada. Neste cenário a *Stream* nunca mais emitirá dado.
2. Depois de alguns dados emitidos a *Stream* é fechada por ocorrer algum erro ao enviar (*Push*) os dados pelo canal.
3. No terceiro cenário a *Stream* emitirá dados pelo canal infinitamente. Este é o cenário de interesse deste trabalho.

Com o foco no terceiro cenário visível acima, deve-se adaptar a aplicação de maneira que conforme os dados estejam disponíveis, ela consiga processá-los

individualmente, e possivelmente gerar novas *Streams* que servirão como *inputs* para outras partes da aplicação. Este conceito é chamado de *pipeline*, que é a composição de funções que recebem *inputs* e geram *outputs* a partir delas, que são executadas sequencialmente, formando um fluxo contínuo de processamento de uma parte a outra da aplicação (FOWLER, 2015). Na Figura 12 abaixo segue um exemplo de *pipeline*. Nele, pode-se observar a composição de três funções:

- Primeira função: filtragem de artigos com mais de 1000 palavras
- Segunda função: mapeamento dos artigos para a sua quantidade de palavras
- Terceira função: reduzir a sequência para o somatório de todas as palavras.

Figura 12 - *pipeline*



Fonte: https://martinfowler.com/articles/collection-pipeline

Com o uso de *pipeline*, a mesma função (método) é aplicada a todos os itens da sequência de dados. Assim é possível lidar com fluxos infinitos de dados, uma vez que é não é necessário ter todos os dados disponíveis no mesmo instante na aplicação. Esta é a ideia sob o conceito de *Lazy Evaluation*, muito empregado em linguagens funcionais como Clojure e Haskell (FOWLER, 2015).

Tanto *Stream* quanto *pipeline* são requisitos para o entendimento do presente trabalho porque eles são a base para a Programação Funcional e Reativa (capítulo 3).

3 PROGRAMAÇÃO FUNCIONAL E REATIVA

Com o aumento contínuo da complexidade das aplicações há necessidade de melhores meios para se organizar e arquitetar o *software* de forma que a sua manutenção não seja uma tarefa árdua. A programação Funcional e Reativa tem como um de seus propósitos facilitar o desenvolvimento de módulos com responsabilidades próprias, em que nenhum outro módulo do sistema pode alterar o seu estado sem ser ele mesmo (CYCLE.JS, 2016).

Este novo padrão de desenvolvimento sofreu influência do chamado *Observer Pattern*. A programação funcional e reativa estende este padrão de desenvolvimento fazendo com que os eventos que cheguem aos *Subscribers* possam ser tratados como *Streams*, que representam um fluxo contínuo de eventos através de um canal.

Um programa implementado sob a ótica da PFR se torna mais declarativo. Ele descreve o que o programa deve fazer, e não como fazer. Há preocupação em como as *Streams* se relacionam, se transformam e se combinam para formar novas *Streams*. Desta forma, diferentemente de programas essencialmente imperativos, onde percebe-se uma riqueza grande de detalhes de implementação, com a PFR o objetivo é de descrever o programa em um nível mais alto de abstração (BLACKHEATH, JONES, 2015, p. 22). A seguir são apresentados os conceitos e definições das estruturas na PFR.

3.1 DEFINIÇÕES

3.1.1 STREAM

Este conceito é de suma importância para o entendimento da Programação Funcional e Reativa. Com a PFR, deixa-se de ver a aplicação como chamadas pontuais de métodos, para ver o programa como diversas sequências de dados (*Stream*) provenientes de diferentes partes da aplicação (COUCHBASE, 2015). Estes dados podem vir de uma chamada Ajax, de uma leitura de registros do banco de dados através de um servidor Node JS, mensagens que chegam à aplicação via conexão *WebSocket*, os *clicks* de um *mouse* sobre um elemento de uma página *Web*, entre outros. A seguir será apresentada uma das definições de *Stream* retirada do livro Functional Reactive Programming (2015):

> *Stream* - uma *stream* de eventos discretos. Também conhecida em outros sistemas de PFR como evento (no qual as coisas que são contidas pela *stream* são chamadas de ocorrências), *stream* de eventos, *observable* ou *signal*. Quando um evento é propagado através de uma *stream*, às vezes dizemos que a *stream* foi disparada (MANSILLA, 2015, tradução livre).

Dentre as diversas bibliotecas, frameworks que implementam este conceito, seja em Java ou Javascript, mas não restrito a estas linguagens, o objetivo é fornecer ferramentas capazes de abstrair a fonte dos dados e prover uma interface em que o modo como se interage com essas *Streams* é unificado, não importando a origem e o tipo dos dados que trafegam por elas. (EXTENSIONS, 2016).

Através desta interface o desenvolvedor consegue se desvincular das especificidades dos dados com os quais está lidando, e começar a pensar no lado conceitual de como as diferentes *Streams* se relacionam e se transformam criando novas *Streams* que darão continuidade ao fluxo de execução do programa (BLACKHEATH, JONES, 2015, p. 22).

Criação de *Stream*

Para as linguagens que possuem bibliotecas que suportam recursos da Programação Funcional e Reativa como Java, JavaScript, Elm, Clojure, entre outros, há um conjunto de métodos disponíveis para a criação de *Streams*. Dentre elas, as que suportam desenvolvimento *front-end Web*, por exemplo, é possível criar diferentes *Streams* a partir de eventos do DOM (*Document Object Model*), como o clique sobre um botão, ou o ato de arrastar um elemento de um lado para outro da tela. Cada evento que ocorre, como o clique do mouse, é interpretado como um novo item da *Stream*.

Transformação de *Stream*

Diversas operações podem ser aplicadas a uma *Stream*. Por exemplo, pode-se ter uma *Stream* que originalmente represente novos *Tweets* publicados a respeito

de um tema. Conforme os *Tweets* são publicados ao longo do tempo, eles se tornam disponíveis na *Stream* implementada pelo *software* em questão. A partir deste momento aplicam-se métodos às *Streams* contendo as regras de negócio da aplicação que irão moldar o fluxo da *Stream*. Abaixo seguem exemplos de passos que poderiam ser aplicados sequencialmente a essa *Stream*:

1. transformar o formato do *Tweet* que veio do servidor de XML (Extensible Markup Language) para JSON (JavaScript Object Notation)
2. filtrar os *Tweets* da *Stream* que tenham menos de 50 caracteres
3. identificar e remover links presentes nos *Tweets*
4. transformar cada *Tweet* em um elemento HTML Table Row (TR), que representa a linha de uma tabela de *Tweets*
5. inserir cada elemento HTML TR na tabela de *Tweets* da interface
6. notificar o usuário da aplicação que novos *Tweets* chegaram

Ao final do processamento de cada passo visto acima uma nova *Stream* é produzida. Variadas transformações podem ser realizadas tanto sobre dados remotos à aplicação, como no exemplo de *Tweets* citado anteriormente, como também internamente na própria aplicação. Alguns outros exemplos de transformações de *Stream* de dados são: *Stream* de vídeo, decodificador de textos em diferentes formatos, conversor dos formatos *Comma-separated Values* (CSV) para JSON, entre outros (DENICOLA; YOSHINO, 2016).

3.1.2 OPERAÇÕES LINQ

A empresa Microsoft criou um componente que abstrai diversas operações que são comuns na manipulação de sequência de dados. A concepção desse componente teve como objetivo unificar a forma como as operações de filtro, *merge* (fusão entre duas sequências de dados), ordenação, agrupamento, entre outras, fossem feitas. O trecho abaixo, extraído da documentação *Language Integrated Query* (LINQ) da Microsoft "*Language Integrated Query* (LINQ)", destaca resumidamente o grande benefício que este componente traz aos desenvolvedores de *software*:

> Para um desenvolvedor que escreve consultas, a parte mais visível de uma “linguagem-integrada” são as expressões LINQ. As expressões de consulta são escritas com uma sintaxe declarativa. Usando esta sintaxe, é possível executar operações de filtro, ordenação e agrupamento em fontes de dados com o mínimo de código. São utilizadas as mesmas expressões de consulta para transformar dados em bases SQL, sequências de dados ADO .NET (MSDN, 2017), documentos XML, *Streams* e coleções .NET. (MICROSOFT, 2017, tradução livre).

É comum a implementação desta interface que unifica as operações em sequências de dados nas bibliotecas/frameworks que dão suporte à PFR. Esta é mais uma evidência da intenção da PFR prover meios que viabilizem a programação declarativa, que tem o foco em expressar o que deve ser feito.

3.1.3 PROGRAMAÇÃO FUNCIONAL COMBINADA À PROGRAMAÇÃO REATIVA

Na seção 2.2 deste trabalho foram explicados alguns dos principais conceitos a respeito do paradigma de Programação Funcional. Tendo esse e os outros assuntos apresentados até o tópico atual em vista, agora é o momento de abordar o encontro das ideias que envolvem a PF, a Programação Reativa e o conceito de *Streams*, que são a base para a PFR.

Programas que seguem a Programação Reativa são expressos através da definição da relação entre os *inputs* do sistema, e quais eventos, áreas do código são ativadas quando determinados eventos são propagados. Diz-se que o programa reage às *inputs*. A programação Funcional e Reativa trabalha com esta mesma abstração, o chamado *Observer Pattern*, onde de um lado existem as entidades que produzem eventos, e do outro as entidades que ficam “escutando” (esperando) o disparo destes eventos para que elas possam reagir, são os chamados *Listeners* ou *Subscribers*. Apesar desta similaridade entre os paradigmas, a PFR trabalha com a ideia de *Stream*, já visto anteriormente, que permite o desenvolvedor utilizar funções (métodos) que poderão ser aplicados às *Streams*.

O motivo pelo qual a palavra Funcional está presente na PFR se dá pelo fato dos métodos (funções) que são aplicados às *Streams* respeitarem as regras da Programação Funcional. Abaixo são apresentadas algumas delas (BLACKHEATH, JONES, 2015, p. 36):

- não é permitido realizar operações de entrada e saída
- não é permitido lançar exceções, a menos que elas sejam tratadas dentro da própria função
- não é permitida a utilização de variáveis declaradas fora do escopo da função
- não é permitido modificar o estado externo à função
- não é permitido manter nenhum estado entre as chamadas das funções

Em uma visão geral, essas funções não devem sofrer ou praticar *side effects* fora do seu escopo. Ou seja, elas devem ser funções puras (BLACKHEATH, JONES, 2015, p. 36).

3.1.4 PARALELISMO

As *Streams*, no contexto da Programação Funcional e Reativa, devem ser imutáveis. Todo o processamento que é feito sobre uma *Stream* ocorre de item a item, ou seja, cada elemento presente no fluxo de dados é independente um do outro assim como também não sofrerá *side effects* de outras partes da aplicação. Esta característica traz benefícios em casos em que faz sentido o processamento paralelo do trabalho a ser feito, utilizando múltiplas *Threads* para distribuí-lo (FOWLER, 2015). A execução de tarefas em paralelo do trabalho é essencial no contexto de aplicações ricas em interação com o usuário, pois o *front-end* dessas aplicações é inerentemente assíncrono (UNIVERSITY, 2016).

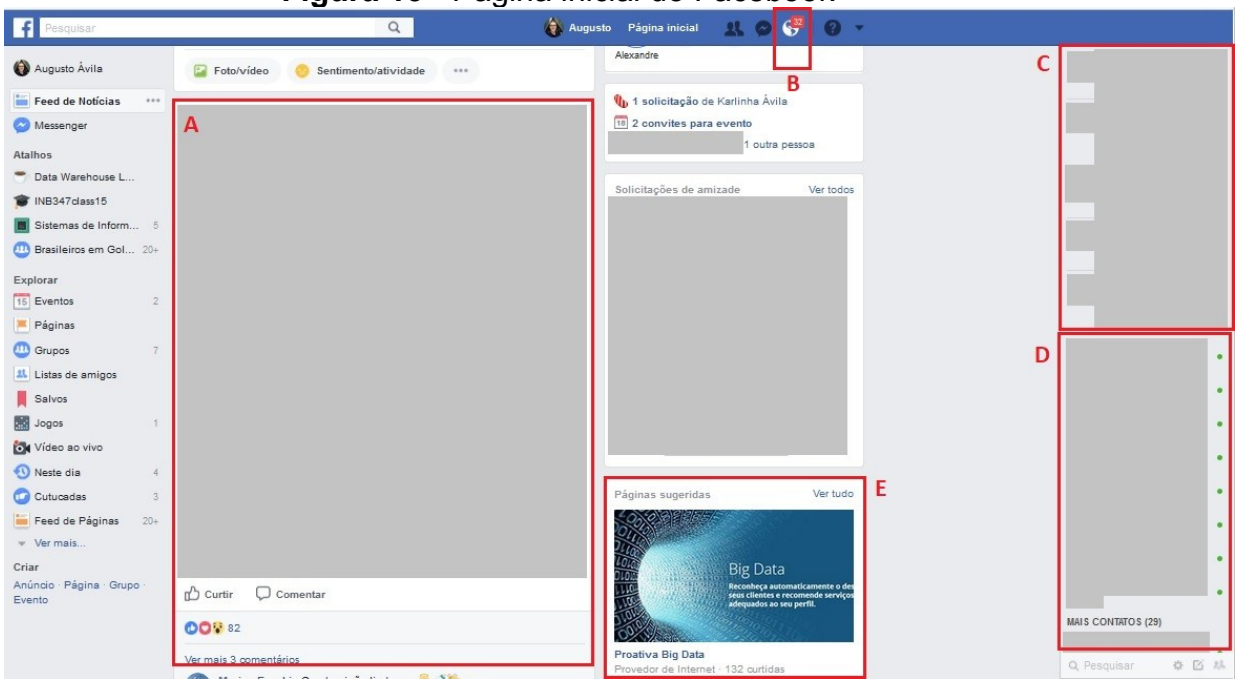
3.2 EXEMPLO DE CONSTRUÇÃO DE INTERFACE - FACEBOOK

Como já apresentado neste trabalho, é crescente o número de aplicações *Web* que trazem informações em tempo real e permitem uma interação dinâmica

com o usuário. Exemplos desse tipo de aplicação são *Dashboards* de gestão com indicadores que são constantemente atualizados, facilitando assim a tomada de decisão por quem o usa; *sites* de compra e venda, como *Amazon* e *Ebay* onde aparecem informações como produtos recomendados, avaliações de outros compradores, alertas de promoções; e redes sociais como *Facebook* e *Twitter*, onde os dados são volumosos e constante através de novas publicações, compartilhamentos e *chats*. São apresentados nesta seção alguns benefícios que a PFR trás para o desenvolvimento do *front-end* desse tipo de aplicação, na qual se faz necessário muita interação entre o *front-end* e *back-end* da aplicação para prover esta rica experiência para o usuário. Será dado como base para esta explicação a página inicial do *Facebook*, na sua versão *Web* (exibido no navegador de páginas *Web* Firefox).

A Figura 13 abaixo representa a página inicial do *Facebook*, segmentada por área. As informações dos usuários foram removidas por questões de privacidade.

Figura 13 - Página inicial do Facebook



Fonte: http://www.facebook.com

Segue explicação de cada área destacada na Figura 13 acima:

A - Linha do Tempo: Esta é uma lista vertical de publicações dos seus amigos. Sempre que um amigo seu faz uma nova publicação ou compartilha um conteúdo

este aparece para você. Conteúdos mais recentes e em destaque ficam no topo na lista.

B - Notificações: Esta área tem um número que é incrementado conforme novas notificações/eventos surgem. Um exemplo de notificação é quando um amigo seu faz um comentário em alguma publicação sua.

C - Atividades Recentes: Esta área é uma lista vertical resumida das atividades recentes dos seus amigos. Por exemplo, uma nova linha aparece no todo desta lista quando um amigo seu curte uma publicação.

D - Chat Online: Esta área apresenta em uma lista vertical os seus amigos que estão disponíveis para conversar no momento.

E - Conteúdo patrocinado: Esta área apresenta conteúdos patrocinados que sejam relevantes para você. De acordo com o conteúdo que você pesquisa, lê e assiste o *Facebook* muda o conteúdo nesta área.

Não é de conhecimento do autor deste trabalho quais são as tecnologias e estratégias utilizadas para o desenvolvimento do *front-end* do Facebook na sua versão para *Web*. Entretanto, será apresentado o Facebook como exemplo uma vez que esta é a rede social mais usada no mundo, e portanto, facilita a contextualização do uso da PFR (DRUM, 2017).

Segundo Roberts (2013), consultor de arquitetura *front-end*, uma das premissas para o desenvolvimento de aplicações *Web* é a preocupação com a velocidade com que os conteúdos são disponibilizados na tela. Como apresentado na seção Protocolo HTTP (2.1.3), em toda requisição que o *front-end* faz para o *back-end* há um arquivo como resposta. Quanto menor for o tamanho deste arquivo, mais rápido será o carregamento da aplicação na tela para o usuário. Uma das estratégias de otimização utilizadas por desenvolvedores *front-end* é minimizar ao máximo o número de requisições ao servidor, mas quando for necessário fazê-las, utilizar recursos de paralelização de requisições para minimizar o tempo de carregamento do conteúdo desejado.

Em aplicações como o *Facebook*, que são segmentadas em diversas áreas onde cada uma pode ser considerada independente das outras, é possível fazer o carregamento do conteúdo das áreas que trazem maior valor para os usuários com maior prioridade. Segue abaixo um exemplo de requisito de *software* que poderia ser adotado pelo *Facebook* e outras aplicações afins, em relação a prioridade no carregamento dos conteúdos da aplicação para o usuário:

Requisito: O conteúdo da Linha do Tempo (área A) é o mais relevante para os usuários, e portanto deve ser o primeiro a ser carregado na interface. As outras áreas têm o mesmo nível de prioridade entre si. Desta forma, não é relevante a ordem de carregamento do conteúdo entre estas áreas de mesma prioridade.

3.2.1 IMPLEMENTAÇÃO

Abaixo é apresentado uma parte da implementação do requisito definido acima. Este exemplo foi desenvolvido com o uso da linguagem de programação *Javascript*, e da biblioteca RxJS (EXTENSIONS, 2016), que é uma das ferramentas disponíveis atualmente no mercado para se trabalhar com os conceitos da PFR no *front-end*. Mais detalhes sobre esta biblioteca são vistos na seção 3.6 - Bibliotecas e *Frameworks*.

Assume-se que o *back-end* da aplicação provê uma API REST, com URIs específicas para cada recurso/área deste exemplo. Mais detalhes sobre a comunicação entre a aplicação cliente e servidora podem ser vistos na seção 2.1, Arquitetura cliente-servidor.

Não será apresentado nas figuras abaixo todo o código relativo a este exemplo. Destaca-se, no entanto, apenas aquelas que demonstram como o fluxo de execução das requisições pode ser coordenado com o uso da PFR.

O código da Figura 14 faz a declaração dos *endpoints* que serão utilizados para consultar os recursos necessários para carregar o conteúdo de cada área da aplicação.

Figura 14 - Exemplo de PFR - Declaração de *endpoints*

```
/*
  Definições dos endpoints do back-end
*/
const ENDPOINT_LINHA_DO_TEMPO       = '/linha_do_tempo';
const ENDPOINT_NOTIFICACOES         = '/notificacoes';
const ENDPOINT_ATIVIDADES_RECENTES  = '/atividades_recentes';
const ENDPOINT_CHATS                = '/chats';
const ENDPOINT_CONTEUDO_PATROCINADO = '/conteudo_patrocinado';
```

Fonte: Elaborado pelo autor

Na Figura 15 faz-se a declaração dos *Subscribers*. No contexto deste exemplo, esses *Subscribers* são os responsáveis por alterar o código HTML da página *Web*. Eles ficam “esperando” eventos chegar até eles, para então “reagir” ao evento produzindo um *side-effect* que é a atualização dos elementos visuais da aplicação.

O elemento *$domReady* é a declaração de uma *Stream* que emitirá apenas um único valor. Quando a estrutura básica do HTML for carregada na tela, será emitido um evento por esta *Stream*. Essa *Stream* é o ponto inicial de execução deste exemplo.

Figura 15 - Exemplo de PFR - Declaração de *Subscribers*

```
var $domReady                    = Rx.DOM.ready();
var linhaDoTempoSubscriber       = new LinhaDoTempoSubscriber();
var notificacoesSubscriber       = new NotificacoesSubscriber();
var atividadesRecentesSubscriber = new AtividadesRecentesSubscriber();
var chatsSubscriber              = new ChatsSubscriber();
var conteudoPatrocinadoSubscriber = new ConteudoPatrocinadoSubscriber();
var linhaDoTempoCarregadaSubject = new Rx.Subject();
```

Fonte: Elaborado pelo autor

A Figura 16 abaixo deixa evidente como a PFR trabalha com fluxo de dados, as chamadas *Streams*. Aqui os dados emitidos por uma *Stream* são mapeados para uma outra *Stream*. Ou seja, os dados são formatados conforme a necessidade do problema.

Ao final da sequência de transformação das *Streams* encontra-se uma chamada ao método *.subscribe()*. Este método recebe o objeto linhaDoTempoSubscriber como parâmetro.

Resumidamente, a Figura 16 mostra que assim que o HTML básico da página for carregado será iniciada a requisição ao *endpoint* responsável por trazer o conteúdo da linha do tempo. Assim que o conteúdo for recebido ele será inserido na área correspondente da interface.

Figura 16 - Exemplo de PFR - Transformação de *Streams*

```
/*
  Quando a estrutura base do HTML for carregada será
    disparado um evento que ativa a requisição para
    consulta de conteúdo da Linha do Tempo
*/
$domReady
  /*
    DOM está pronto, então agora é hora de consultar
      a Linha do Tempo no backend
  */
  .map(domReady => Rx.DOM.Request.get(ENDPOINT_LINHA_DO_TEMPO))
  /*
    Conteúdo da Linha do Tempo chegou
  */
  .map(linhaDoTempo => {
    /*
      Emitir evento que a linha do tempo foi carregada
    */
    $linhaDoTempoCarregadaSubject.onNext(true);
    $linhaDoTempoCarregadaSubject.onCompleted();
    return linhaDoTempo;
  })
  /*
    Atualizar a área da interface relacionada ao
      conteúdo da linha do tempo
  */
  .subscribe(linhaDoTempoSubscriber);
```

Fonte: Elaborado pelo autor

A Figura 17 abaixo mostra a dependência entre o conteúdo da linha do tempo ter sido carregado e as requisições para buscar os conteúdos das outras áreas de menor prioridade para a aplicação. Este é um dos objetivos da PFR, que é facilitar a declaração do que o *software* deve fazer.

Percebe-se neste exemplo que não foi necessário mostrar o código responsável por atualizar o HTML das áreas da interface para que o leitor entenda o que o *software* se propõe a fazer. Esta é uma demonstração de como o *software* pode ser desenvolvido com componentes com responsabilidades bem definidas, e de forma declarativa.

Figura 17 - Exemplo de PFR - Declaração de dependência entre segmentos da aplicação

```
/*
  Logo que a linha do tempo for carregada faremos as próximas requisições
    para buscar os conteúdos das outras áreas
*/
$linhaDoTempoCarregadaSubject
  .subscribe(linhaDoTempoCarregada => {
    /*
      Essas requisições abaixo serão feitas em paralelo. A interface
        das áreas serão atualizadas independentemente umas das outras
    */
    Rx.DOM.Request.get(ENDPOINT_NOTIFICACOES)
      .subscribe(notificacoesSubscriber);

    Rx.DOM.Request.get(ENDPOINT_ATIVIDADES_RECENTES)
      .subscribe(atividadesRecentesSubscriber);

    Rx.DOM.Request.get(ENDPOINT_CHATS)
      .subscribe(chatsSubscriber);

    Rx.DOM.Request.get(ENDPOINT_CONTEUDO_PATROCINADO)
      .subscribe(conteudoPatrocinadoSubscriber);
  });
```

Fonte: Elaborado pelo autor

3.6 BIBLIOTECAS E FRAMEWORKS

Com tantos desafios associados ao desenvolvimento do *front-end* de aplicações *web*, novos meios de gerenciar a complexidade destas aplicações foram criados. Algumas dessas tecnologias são Redux (REDUX, 2016), Flux (FLUX, 2017), React (REACT, 2017), Vue JS (VUE, 2017), Angular (ANGULARJS, 2017), Cycle JS (CYCLEJS, 2017) e Elm (CZAPLICKI, 2017), e todas de alguma forma ajudam os desenvolvedores a abstrair a complexidade da resolução dos problemas do *front-end* de aplicações altamente dinâmicas. Outra tecnologia que tem grande atenção atualmente é o chamado *Reactive Extensions*. Embora esta seja uma ferramenta, e não um *framework*, ela teve grande influência sobre o tema deste trabalho. A seguir ela e alguns outros frameworks são apresentados em mais detalhes.

3.6.1 REACTIVE EXTENSIONS

Reactive Extensions, também conhecido como ReactiveX ou simplesmente Rx, é uma API que permite programação assíncrona utilizando *Observable Streams*. Foi inicialmente desenvolvida pelo arquiteto Eric Meijer e seu time na Microsoft para a linguagem .NET (MICROSOFT, 2012). Posteriormente, com o ganho de popularidade, esta API foi também portada para diversas outras linguagens, como: Java, Javascript, Scala, Swift, entre outras (REACTIVEX, 2017). Esta é a principal biblioteca atualmente utilizada quando se deseja utilizar os conceitos da Programação Funcional e Reativa em prática. Alguns são os possíveis motivos para sua popularidade, dentre eles:

- vasta documentação, com diversos exemplos e linguagem de fácil entendimento;
- comunidade ativa;
- grande número de funcionalidades de alto nível que abstraem problemas complexos na programação assíncrona;
- uso desta biblioteca em outros grandes projetos como Angular, da Google e o *framework* CycleJS.

Em 2012 a Microsoft abriu seu código e desde então a API se popularizou e ficou mais robusta em termos de funcionalidade e confiabilidade uma vez que grandes empresas, como Google e Netflix dão suporte e a utilizam em seus produtos. Segundo o próprio líder do projeto, Eric Meijer, “Rx é a cura para o código assíncrono spaghetti” (MICROSOFT, 2012).

3.6.2 ANGULAR JS

Angular JS é um *framework* de código aberto desenvolvido e mantido pela Google para se trabalhar com aplicações *web* dinâmicas. Ele permite o uso do HTML como sua linguagem de template e assim permite também a extensão da sua sintaxe para expressar os componentes da aplicação de forma simples e sucinta (ANGULARJS, 2017).

A primeira versão deste framework, o Angula JS 1, não foi concebido com o foco em mobilidade, ou seja, dispositivos móveis, e umas das consequências disso foi o baixo desempenho na execução destas aplicações. Apesar da baixa performance, o *framework* teve grande aceitação pelos desenvolvedores (INFOWORLD, 2015).

Para corrigir este problema de eficiência, a Google lançou uma nova versão do framework, chamado Angular 4, o qual pode ser interpretado como uma completa reescrita do *framework*, uma vez que diversos novos conceitos foram empregados, mas o que chama mais atenção para o presente trabalho é o uso de *Streams* e *Observables* na estrutura básica do Angular 4 (INFOWORLD, 2015).

Esta nova versão do *framework* utiliza a biblioteca RxJS, que é a versão da Reactive Extensions para a linguagem de programação Javascript, na sua estrutura básica. São 3 as áreas onde o uso dos *Observables* mais se faz presente: EventEmitter, HTTP e Forms (ANGULARJS, 2017). Como já visto neste trabalho, um dos objetivos da PFR é tornar o *software* mais declarativo e facilitar sua manutenção. Segue abaixo uma frase retirada da introdução da documentação do guia de desenvolvimento do Angular que faz sentido com o assunto presente:

> AngularJS foi construído com a crença de que código declarativo é melhor que imperativo quando o assunto é o desenvolvimento de Interfaces de Usuário e a ligação entre componentes de *software*,

> enquanto que código imperativo é excelente para expressar regras de negócio (ANGULARJS, 2017, tradução livre).

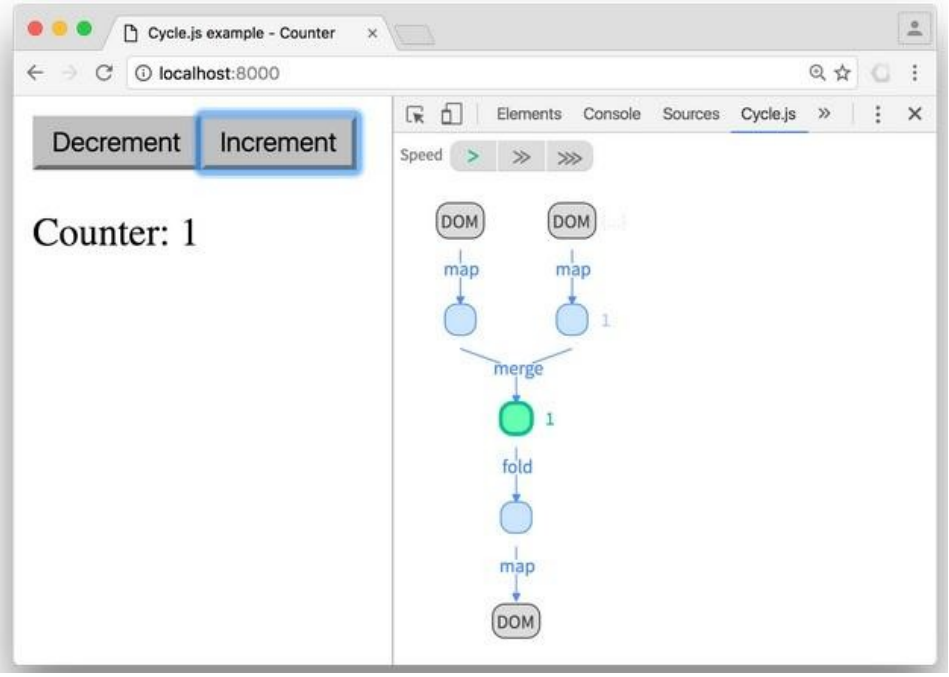
3.6.3 CYCLE JS

A mensagem em destaque no seu site oficial descreve em poucas palavras o que a ferramenta é: “Um *framework* Javascript funcional e reativo para código previsível” (CYCLE.JS, 2017, tradução livre). Este *framework* foi desenvolvido inicialmente por André Staltz, mas atualmente tem suporte da comunidade Cycle, que vem crescendo ano a ano, sendo que encontros anuais são feitos para reunir entusiastas e apresentar a evolução do framework, como o evento Cycle Conf 2017. Toda a base da sua estrutura e conceitos utilizados na sua concepção são fundamentados no conceito de *Streams*, como já abordado neste trabalho.

O site oficial destaca algumas de suas características:

- **funcional e reativo**. Um código escrito seguindo os princípios da programação funcional se torna previsível. Já o conceito de código reativo está relacionado à separação de responsabilidades.
- **simples e conciso**. O autor destaca que o *framework* é simples e com poucos conceitos a serem aprendidos para utilizá-lo.
- **extensível e testável**. O *framework* permite facilmente adicionar *plugins* à aplicação. Pelo fato de toda a aplicação ser desenvolvida com programação funcional, a inspeção e teste do código se tornam simples.
- **fluxo de dados explícito**. Diferentemente de aplicações imperativas onde o desenvolvedor que deseja entender o fluxo dos dados deve construir um mapa mental da execução do código, com o Cycle JS os fluxos dos dados são declarados explicitamente no código. Além desta facilidade, a comunidade Cycle desenvolveu uma ferramenta que ajuda o desenvolvedor a visualizar a execução dos fluxos da aplicação conforme eles são ativados. A Figura 18 abaixo exemplifica uma aplicação com dois botões. Quando cada um deles é clicado, uma *Stream* é ativa. Sua execução é visualizada no painel à direita da tela:

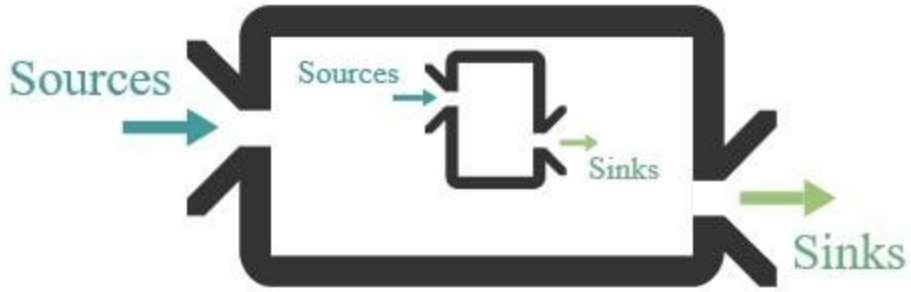
Figura 18 – Fluxo de Dados



Fonte: https://cycle.js.org/

- **composição**. Este framework facilita a reutilização de componentes. É destacado no site oficial do Cycle que, diferentemente de outros frameworks, toda aplicação Cycle pode ser utilizada como um componente, pois todas seguem o mesmo padrão de entradas e saídas, assim como uma função que pode ser utilizada em outra aplicação mais complexa. A Figura 19 abaixo ilustra o uso de componentes.

Figura 19 – Componentes Cycle



Fonte: https://cycle.js.org/

3.7 ANÁLISE DOS FRAMEWORKS QUANTO AO USO DA FERRAMENTA RX E DA TÉCNICA PFR

A ferramenta Rx, disponível em inúmeras linguagens de programação, vem se popularizando com o seu uso no desenvolvimento do *front-end* de aplicações *web* com sua versão para *Javascript*, o RxJS.

O Angular JS, como já mencionado anteriormente, foi reescrito para sua mais atual versão, Angular 4, que possibilita a aplicação de conceitos da Programação Funcional e Reativa por meio do RxJS. O *framework* Angular se mostra flexível suficiente para que o desenvolvedor consiga escolher onde deseja aplicar a PFR em sua aplicação. Por exemplo, é possível fazer requisições HTTP ao servidor recebendo um *Observable (Stream)* como retorno, ou então fazê-lo do modo tradicional com uma requisição AJAX, recebendo um *callback* como retorno. Também é viável a concepção completa de uma aplicação Angular fazendo intenso uso da Programação Funcional e Reativa. Neste caso, a biblioteca RxJS será utilizada frequentemente (UNIVERSITY, 2016).

O *framework* Cycle JS, diferentemente do Angular JS, impõe uma arquitetura mais rígida quanto ao seu uso. Aplicações desenvolvidas em Cycle JS devem ser concebidas fazendo uso constante dos *Observables*, e consequentemente pouco ou nenhum estado é manuseado diretamente pelo código do desenvolvedor. O estado da aplicação fica armazenado e isolado de mudanças internamente nos *Observables*, diminuindo assim as chances de ocorrerem *side effects* e *bugs*. Por fim, em relação ao uso da biblioteca RxJS, é esperado que se faça uso intenso desta ferramenta para viabilizar a implementação dos conceitos da Programação Funcional e Reativa.

A seguir é apresentada uma análise comparativa entre os *frameworks* Angular JS e Cycle JS quanto ao uso da técnica PFR e da ferramenta Rx.

Tabela 1 – Comparação entre os *Frameworks* Angular JS e Cycle JS

Característica	Angular JS	Cycle JS
Arquitetura flexível quanto ao uso da PFR	Sim	Não
Flexibilidade quanto ao uso da ferramenta RxJS	Sim	Não
Possibilidade de uso de *Observables*	Restrito a poucos módulos	Possível utilizar em todo o *framework*
Gerenciamento do estado da aplicação	Através de variáveis de contexto e *Observables*	*Observables*

Fonte: elaborado pelo autor

4 CONCLUSÃO

O presente trabalho explorou o tema PFR como uma possível alternativa que pode ser adotada em projetos que tenham a característica de alta complexidade no desenvolvimento do *front-end* de aplicações *web*.

Diversas são as tecnologias e opiniões a respeito de como deve ser desenvolvida e estruturada esta camada da aplicação, e ainda não há um modelo adotado por todos, assim como não há uma tecnologia que pode ser utilizada em todos os contextos de aplicações ("*one size fits all*"). O que foi enfatizado ao longo deste projeto é a aderência desta técnica de programação para o desenvolvimento do *front-end* de aplicações interativas, dinâmicas.

As conclusões que foram obtidas a respeito da técnica PFR tiveram base nas referências bibliográficas presente neste trabalho e também através da experiência prática que o autor teve ao utilizar a PFR no seu dia-a-dia como desenvolvedor de *software*.

Verificou-se que a técnica de PFR facilita a leitura e interpretação do código pelo desenvolvedor por permitir que a implementação seja feita em um grau de abstração maior que as técnicas utilizadas atualmente, como o uso de *callback* e *promises*. Além deste benefício, a PFR incentiva o uso de componentes com responsabilidades específicas, que ficam isolados de outras partes da aplicação. Essas e outras características desta técnica promovem a qualidade do *software* e facilitam a sua manutenção.

Apesar do aumento da qualidade do *software* com o uso da PFR, alguns pontos podem ser interpretados como negativos quanto ao seu uso. Verificou-se que é longo o período de aprendizado por parte do desenvolvedor, poucas ferramentas que facilitem a inspeção do *software* em execução e o grande número de linhas de código, por mais que seja utilizada uma ferramenta própria para o desenvolvimento, como a RxJS (própria para Javascript). Futuramente, com a evolução das linguagens de programação, possivelmente haja suporte nativo de instruções que simplifiquem o uso da PFR, deixando o código mais conciso, limpo e declarativo.

Além das limitações citadas anteriormente, também se observou pouco material disponível para um aprofundamento prático neste tema, e isso pode ser

uma barreira para o progresso nos estudos do desenvolvedor. Sugere-se que, para aplicações que já existam e nas quais se deseja adotar estes novos conceitos, se utilize a FPR em partes isoladas da aplicação. Conforme ganha-se mais confiança, o desenvolvedor pode adotar esta prática em outros segmentos de código, e assim por diante.

Por ser considerada uma nova abordagem na resolução de problemas do *front-end* e fazer com que o desenvolvedor tenha que desconstruir seu modelo tradicional de desenvolvimento de *software* imperativo (padrão atual), a PFR possivelmente será adotada de forma mais significativa pelos desenvolvedores no decorrer dos próximos anos. O desenvolvedor levou anos para ter confiança com a Programação Funcional, e levou anos para ter confiança com a Programação Orientada à Objetos. É provável que o mesmo ocorra com a Programação Funcional e Reativa.

Sugere-se como trabalho futuro o estudo da PFR aplicado à outra camada da aplicação, o *back-end*. A empresa Netflix está a frente deste tópico e adotou a PFR nas suas APIs como um meio de paralelizar as requisições ao seu *back-end*, diminuindo assim o tempo de requisição e facilitando a manutenção do seu código (CHRISTENSEN; HUSAIN, 2014). É necessário explorar e analisar o uso da PFR em outros cenários de *back-end*, fazendo um levantamento dos desafios e possíveis melhorias na aplicação com o seu uso.

5 REFERÊNCIAS

ANGULARJS**. What Is AngularJS?** 2017. Disponível em: <https://docs.angularjs.org/guide/introduction>. Acesso em: 22 maio 2017.

BARKLEY, R. J. **Highlights of the History of the Lambda-Calculus**. Annals of the History of Computing, v.6, n. 4, p. 33,. 1984.

BLACKHEATH, S; JONES, A**. Functional Reactive Programming**. Greenwich: Manning Publications, 2015. 320 p.

BONÉR, J. et al. **O Manifesto Reativo**. 2014. Disponível em: <http://www.reactivemanifesto.org/pt-BR>. Acesso em: 02 maio 2016.

CALLBACKHELL. **Callback Hell**: A guide to writing asynchronous JavaScript programs. 2016. Disponível em: <http://callbackhell.com/>. Acesso em: 25 out. 2016.

CHRISTENSEN, Ben; HUSAIN, Jafar. **Reactive Programming in the Netflix API with RxJava**. 2014. Disponível em: <https://medium.com/netflix-techblog/reactive-programming-in-the-netflix-api-with-rxjava-7811c3a1496a>. Acesso em: 05 jun. 2017.

CLARKE, Steven**. Measuring API Usability.** 2004. Disponível em: <http://www.drdobbs.com/windows/measuring-api-usability/184405654>. Acesso em: 02 out. 2017.

COUCHBASE**. Mastering observables**. 2015. Disponível em: <http://docs.couchbase.com/developer/java-2.0/observables.html>. Acesso em: 15 dez. 2015.

CYCLE.JS. **Cycle.js: A functional and reactive JavaScript framework for predictable code**. 2017. Disponível em: <https://cycle.js.org/>. Acesso em: 02 jun. 2017.

CYCLE.JS. **Streams: Reactive Programming**. 2016. Disponível em: <http://cycle.js.org/streams.html>. Acesso em: 18 set. 2016.

CZAPLICKI, Evan. **Elm:** A delightful language for reliable webapps. 2017. Disponível em: <http://elm-lang.org/>. Acesso em: 20 out. 2017.

DENICOLA, D.; YOSHINO, T. **Streams**: Living Standard. 2016. Disponível em: <https://streams.spec.whatwg.org/>. Acesso em: 22 out. 2016.

D'AMORE, M. J.; OBERST, D. J. (1983). **Microcomputers and mainframes: A marriage of effectiveness**. Proceedings of the 11th annual ACM SIGUCCS conference on User services - SIGUCCS '83. p. 7. ISBN 0897911164. doi:10.1145/800041.801417.

DESIGN, **Responsive App. Responsive App Design: Resources for building applications for mobile + desktop**. 2013. Disponível em: <http://www.responsiveappdesign.org/about.html>. Acesso em: 28 jul. 2016.

DEVELOPERS, Google. **JavaScript Promises**: an Introduction. 2016. Disponível em: <https://developers.google.com/web/fundamentals/getting-started/primers/promises>. Acesso em: 05 out. 2016.

DRUM, Marluci. **As 10 maiores redes sociais - Atualizado**: Confira a lista, atualizada em agosto de 2017, das maiores redes sociais da internet. Facebook, WhatsApp, Messenger e Youtube lideram o ranking das campeãs em números de usuários. 2017. Disponível em: <https://www.oficinadanet.com.br/post/16064-quais-sao-as-dez-maiores-redes-sociais>. Acesso em: 10 set. 2017.

EXTENSIONS, R. **The Reactive Extensions for JavaScript (RxJS) 4.0**: About the Reactive Extensions. 2016. Disponível em: <https://github.com/Reactive-Extensions/RxJS>. Acesso em: 01 set. 2016.

FIELDING, R. et al**. Hypertext Transfer Protocol**: HTTP/1.1. 1999. Disponível em: <http://www.rfc-base.org/txt/rfc-2616.txt>. Acesso em: 02 out. 2017.

FIELDING, Roy Thomas**. Representational State Transfer (REST)**. 2000. Disponível em: <http://www.ics.uci.edu/~fielding/pubs/dissertation/rest_arch_style.htm>. Acesso em: 02 out. 2017.

FLUX. **Flux:** APPLICATION ARCHITECTURE FOR BUILDING USER INTERFACES. 2017. Disponível em: <https://facebook.github.io/flux/>. Acesso em: 20 out. 2017.

FOWLER, M. **Collection Pipeline**: Defining Collection Pipeline. 2015. Disponível em: <http://martinfowler.com/articles/collection-pipeline/>. Acesso em: 20 out. 2016.

HINSEN, K. **The Promises of Functional Programming**. IEEE. Computing in Science and Engeneering, v. 11, p.68-75, 2009. Disponível em: <http://ieeexplore.ieee.org/xpl/articleDetails.jsp?arnumber=5076325&newsearch=true&queryText=the%20promises%20of%20functional%20programming>.

HUDAK, P. **The Conception, Evolution, and Application of Functional Programming Languages**. Departamento de Ciência da Computação. Universidade de Yale, mar. 1989. p. 1-87. Disponível em: <http://haskell.cs.yale.edu/wp-content/uploads/2011/01/cs.pdf>.

INFOWORLD. **Google rewrites its Angular JavaScript framework**: Version 2 offers a significant speed boost and improved mobile development capabilities. 2015. Disponível em: <http://www.infoworld.com/article/3015381/javascript/google-rewrites-its-angular-javascript-framework.html>. Acesso em: 22 maio 2017.

LODASH. **Lodash:** A modern JavaScript utility library delivering modularity, performance & extras.. 2017. Disponível em: <https://lodash.com/>. Acesso em: 20 out. 2017.

MACHADO, R. **An Introduction to Lambda Calculus and Functional Programming**. Proceeding of the 2nd WEIT; Rio Grande. 2013 Apr. p. 26–33. Disponível em: <https://www.computer.org/csdl/proceedings/weit/2013/3057/00/3057a026.pdf>. Acesso em: 18 jun. 2016.

MANSILLA, S. **Reactive Programming with RxJS**: Untangle Your Asynchronous JavaScript Code. Dallas: The Pragmatic Programmers, 2015. 121 p.

MEDEIROS, A. C. S. **The introduction to Reactive Programming you've been missing**. 2014. Disponível em: <https://gist.github.com/staltz/868e7e9bc2a7b8c1f754>. Acesso em: 10 mar. 2016.

MEDEIROS, A. **Dynamics of Change**: Why Reactivity Matters. 2016. Disponível em: <http://queue.acm.org/detail.cfm?id=2971330>. Acesso em: 27 ago. 2016.

MEIJER, E. **Your Mouse is a Database**: Web and mobile applications are increasingly composed of asynchronous and realtime streaming services and push notifications. 2012. Disponível em: <http://queue.acm.org/detail.cfm?id=2169076>. Acesso em: 10 abr. 2017.

MICROSOFT. **Guia de F#.** 2017. Disponível em: <https://docs.microsoft.com/pt-br/dotnet/fsharp/>. Acesso em: 22 out. 2017.

MICROSOFT, Docs. **Language Integrated Query (LINQ)**. 2017. Disponível em: <https://docs.microsoft.com/en-us/dotnet/csharp/linq/>. Acesso em: 20 maio 2017.

MICROSOFT. **MS Open Tech Open Sources Rx (Reactive Extensions) – a Cure for Asynchronous Data Streams in Cloud Programming**. 2012. Disponível em: <https://blogs.msdn.microsoft.com/interoperability/2012/11/06/ms-open-tech-open-

sources-rx-reactive-extensions-a-cure-for-asynchronous-data-streams-in-cloud-programming/>. Acesso em: 20 maio 2017.

MSDN. **ADO.NET:** .NET Framework (current version). 2017. Disponível em: <https://msdn.microsoft.com/pt-br/library/e80y5yhx(v=vs.110).aspx>. Acesso em: 20 out. 2017.

NETWORK, Mozilla Developer. **Functions**. 2016. Disponível em: <https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Functions>. Acesso em: 20 jun. 2016.

O'REILLY. **What Is Web 2.0**: Design Patterns and Business Models for the Next Generation of Software. 2016. Disponível em: <http://www.oreilly.com/pub/a/web2/archive/what-is-web-20.html?page=2>. Acesso em: 10 jun. 2016.

RAMDA. **Ramda:** A practical functional library for JavaScript programmers.. 2017. Disponível em: <http://ramdajs.com/>. Acesso em: 20 out. 2017.

REACTIVEX. **ReactiveX**: An API for asynchronous programming with observable streams. 2017. Disponível em: <http://reactivex.io/>. Acesso em: 20 maio 2017.

REACT. **React:** A JavaScript library for building user interfaces. 2017. Disponível em: <https://reactjs.org/>. Acesso em: 20 out. 2017.

REDUX. **Redux**: Motivation. 2016. Disponível em: <http://redux.js.org/docs/introduction/Motivation.html>. Acesso em: 22 out. 2016.

RICHARDSON, Chris; SMITH, Floyd. **MICROSERVICES: From Design to Deployment**. [s. L.]: Nginx, 2016. 74 p. Disponível em: <https://www.nginx.com/resources/library/designing-deploying-microservices/>. Acesso em: 14 set. 2016.

RISKE, Al. **The Soul of a New Programming Language:** Fortress focuses on the needs of scientists.. 2005. Disponível em: <https://web.archive.org/web/20090416193110/http://research.sun.com/minds/2005-0302/>. Acesso em: 20 out. 2017.

ROBERTS, Harry. **Front-end performance for web designers and front-end developers**. 2013. Disponível em: <https://csswizardry.com/2013/01/front-end-performance-for-web-designers-and-front-end-developers/#section:make-fewer-requests>. Acesso em: 10 out. 2017.

STELOVSKY, J.; ASCHWANDEN, C. Software architecture for unified management of event notification and stream I/O and its use for recording and analysis of user events. In: HAWAII INTERNATIONAL CONFERENCE ON SYSTEM SCIENCES, 35., 2002, Manoa. **Software architecture for unified management of event notification and stream I/O and its use for recording and analysis of user events**. Manoa: Ieee, 2002. p. 1 - 6. Disponível em: <http://ieeexplore.ieee.org/document/994101/?section=abstract>. Acesso em: 20 out. 2016.

VUE. **Vue:** The Progressive JavaScript Framework. 2017. Disponível em: <https://vuejs.org/>. Acesso em: 20 out. 2017.

UNIVERSITY, Angular**. Functional Reactive Programming for Angular Developers - RxJs and Observables**. 2016. Disponível em: <http://blog.angular-university.io/functional-reactive-programming-for-angular-2-developers-rxjs-and-observables/>. Acesso em: 10 jun. 2017.

ZAKAS, Nicholas C.. **Node.js and the new web front-end**. 2013. Disponível em: <https://www.nczonline.net/blog/2013/10/07/node-js-and-the-new-web-front-end/>. Acesso em: 02 out. 2017.